Motivation

• Many data sets can be modeled as time series (i.e. in which every piece of data has an associated time stamp and their ordering according to time stamps is significant in some way), e.g.
  – monitoring data
  – packets sent on a communication channel

• Providing storage of and efficient access to such data sets is important
Service Architecture

• Multiple service instances, running on multiple machines
  • Each instance runs independently of the other instances
  • A service instance acts as a warehouse for a set of files related to a single time series
  • A file stores data related to a single time series (data or index entries)
  • Each time series has a unique identifier (tsid)
  • General information about each time series (identifier, set of file names and numbers, number of data entries, number of index entries at each level, current index entries at each indexing level, etc.) is stored by each instance in main memory (RAM)
Service Architecture

• Periodically, the general information is checkpointed, in order to be able to stop the instance and restart it later (possibly on a different machine), if necessary.

• Each entry of a time series consists of a pair $(timestamp, value)$.

• It is not required for the values of all the entries of a time series to have the same size.

• On the other hand, all the timestamps of all entries of a time series have the same size.

• The (data and index) entries of a time series are numbered with consecutive numbers, starting from 0 (this number is the position of an entry).

• The positions of the data entries correspond to the order in which the entries are added to the time series and continues over multiple data or index files (each file stores entries numbered consecutively).
Each instance provides to the clients the same interface, through which, at any time, a client may:

- create a new time series
- insert one or more consecutive entries at the end of an existing time series (not necessarily in increasing order of timestamps)
- query a time series

Two types of queries:
- retrieve all the entries of a given time series whose timestamps are between $t_1$ and $t_2$ (inclusive), and
- retrieve all the entries of a given time series between positions $p_1$ and $p_2$ (inclusive)

Both types of queries may take some extra options

Periodically, the instances synchronize their data...
Replication Mechanisms

- Let's consider a time series \( tsid \). Entries of this time series are generated periodically and added to the time series storage and retrieval service by a client.

- Assumptions:
  - the client always uses the same service instance for adding new entries to the time series => **prefix replication**
  - The client may use multiple service instances for adding new entries to the time series => **general-case replication**
Prefix Replication (1)

- Periodically, each service instance \( S \) asks from its neighbors for summary information about all the time series they store, like:
  - the time series identifier \( \text{tsid} \)
  - \( \text{num\_entries}(X, \text{tsid}) \) = the number of entries of the series \( \text{tsid} \) stored by the neighbor \( X \)

- After receiving all the answers (ignoring answers which may have timed out), \( S \) creates locally all the time series whose identifiers were not known to \( S \) previously

- Afterwards, \( S \) considers each time series \( \text{tsid} \):
  - Let \( \text{max\_num\_entries}(\text{tsid}) = \max\{\text{num\_entries}(X, \text{tsid}) | X \text{ is a neighbor of } S\} \)
  - While \( \text{num\_entries}(S, \text{tsid}) < \text{max\_num\_entries}(\text{tsid}) \):
    - \( S \) selects a neighbor \( X \) such that \( \text{num\_entries}(X) > \text{num\_entries}(S) \) and asks from it all the entries between the positions:
      - \( p_1 = \text{num\_entries}(S, \text{tsid}) \)
      - \( p_2 = \min\{\text{num\_entries}(S, \text{tsid}) + K, \text{max\_num\_entries}(\text{tsid}) - 1\} \)
Prefix Replication (2)

K is used as an upper bound for the maximum number of entries which are requested at the same time. All the entries received from the neighbor X (in ascending order of their position) are added by S at the end of its local copy of the time series tsid (also updating the appropriate indices).

Note: because all the entries are inserted into the system through only one service instance SI, they will be propagated to the other instances in the same order in which they were inserted at SI. Because of this, it is correct to assume that, if $num\_entries(S,tsid) < num\_entries(X,tsid)$, then the first $num\_entries(S,tsid)$ entries of the local copy of tsid stored by X are identical to the ones of the local copy of S.
General-case replication (1)

• Our previous assumption that any two service instances have common prefixes of the same time series fails now.

• A service instance $S$ asks each neighbor for summary information regarding all the time series – including the $t_{\text{max}}(*,*)$ and $t_{\text{min}}(*,*)$ values defined below.

• Then, $S$ will use a sliding window (which will slide circularly) $[t_1(S,tsid), t_2(S,tsid)]$ for each time series $tsid$.

• Let $t_{\text{max}}(S,tsid)$ = the maximum timestamp of an entry of the time series $tsid$ stored by $S$ (or $-\infty$ if $S$ currently stores no entry of $tsid$).

• Let $t_{\text{max}}(S,tsid)$ = $\max\{t_{\text{max}}(S,tsid), \max\{t_{\text{max}}(X,tsid) \mid X \text{ is a neighbor of } S\}\}$.

• Let $t_{\text{min}}(S,tsid)$ and $t_{\text{min}}(S,tsid)$ have the same meaning, except that we replace maximum by minimum.

• Let $t_{\text{min}}(S,tsid)$ = $+\infty$ if $S$ currently stores no entry of $tsid$.

• If $t_{\text{max}}(S,tsid) > t_{\text{max}}(S,tsid)$ then we set the sliding window to $[t_1(S,tsid) = t_{\text{max}}(S,tsid) - W, t_2(S,tsid) = t_{\text{max}}(S,tsid)]$ – $W$ is a predefined window size (in terms of timestamp intervals), which should ensure that not too many entries are located in such a time window.
General-case replication (2)

- Otherwise, we set: 
  \[ t_2(S,tsid) = t_1(S,tsid) \]
  and then:
  \[ t_1(S,tsid) = t_2(S,tsid) - W \]
- if we get
  \[ t_2(S,tsid) < t_{\text{min}}(S,tsid) \]
  then we set:
  \[ t_1(S,tsid) = t_{\text{max}}(S,tsid) - W \]
  and:
  \[ t_2(S,tsid) = t_{\text{max}}(S,tsid) \]
- First, \( S \) queries itself and obtains the set of all of the entries of the time series \( tsid \) with timestamps between \( t_1(S,tsid) \) and \( t_2(S,tsid) \) currently stored by \( S \), sorted according to timestamp (denoted by \( CSE(S,tsid,t_1(S,tsid),t_2(S,tsid)) \))
- Let \( \text{num} \) be the number of such entries
- Then, for each neighbor \( X \), \( S \) asks the same query (for the time interval \( [t_1(S,tsid),t_2(S,tsid)] \)), except that it instructs the neighbor to return nothing if the neighbor's result does not have more than \( \text{num}' \)
  \( \text{num}' = 0 \) if we care about correct full replication
  \( \text{num}' = \text{num} \) if we want to save bandwidth
General-case replication (3)

• The returned entries (if any) are merged with $CSE(S, tsid, t1(S, tsid), t2(S, tsid))$

• If new entries not stored by $S$ are discovered, they are added at the end of $S$'s local copy of $tsid$ and inserted into $CSE(S, tsid, t1(S, tsid), t2(S, tsid))$

• After updating the entries in the interval $[t1(S, tsid), t2(S, tsid)]$, we modify the endpoints, by decreasing them both.

• Note: unlike the prefix replication case, when, except for periodic summaries, neighbors were queried only when we were sure that $S$ was missing some entries, this process must be performed continuously (as a newly added entry may have any timestamp). To optimize this process, we can state that we only care for the entries whose timestamps are between $TA$ and $TB$. 

...
Testing

• Prefix replication test with 3 service instances (A, B, C) – A and C in Geneva – B in Amsterdam
• A client (running on the same machine as A) continuously inserted entries (with real values) of one time series for 4 minutes using instance A
• Instance B got its data from A
• C obtained its data from both A and B
• Results: – A was able to write data at a speed of approx. 600,000 entries/sec – C replicated data at about 250,000 entries/sec during the first 4 minutes and at about 300,000 entries/sec after no more entries were written to A – B was able to replicate data at a speed of approx. 90,000 entries/sec during the first 4 minutes and 110,000 entries/sec afterwards – We used K = 25,000 – The replication speed difference between B and C was caused by different RTTs towards A